Problem 1:

Approach:

The implementation of the steepest gradient descent method is as described in lecture slide 42 and 43. The 1D optimization algorithm used to calculate α, the linear search parameter, is the Newton's method (the Golden Section search method requires human judgement to determine the appropriate unimodal bracketing interval, which would need to be repeated for each gradient descent iteration.)

The form given for the Newton 1D method is:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

For this application, the independent variable is α_k (that is $x = \alpha_k$). The minimized 1D function is:

$$\phi(\alpha_k) = f\big(\boldsymbol{x_k} - \alpha_k \boldsymbol{\nabla f}(\boldsymbol{x_k})\big)$$

That is: $\phi(\alpha)$ is the $f(x)$ in the standard notation of Newton method. Note that both the vector $\boldsymbol{x_k}$ and the corresponding gradient vector $\boldsymbol{\nabla f}(\boldsymbol{x_k})$ are treated as constants in the context of the 1D optimization problem (even though they vary for different iterations of gradient descent). By chain rule, we have:

$$\phi'(\alpha_k) = (\boldsymbol{\nabla f})^{\boldsymbol{T}}\big(\boldsymbol{x_k} - \alpha_k \boldsymbol{\nabla f}(\boldsymbol{x_k})\big) \cdot \frac{d(\boldsymbol{x_k} - \alpha_k \boldsymbol{\nabla f}(\boldsymbol{x_k}))}{d\alpha_k}$$

$$\phi'(\alpha_k) = -(\boldsymbol{\nabla f})^{\boldsymbol{T}}\big(\boldsymbol{x_k} - \alpha_k \boldsymbol{\nabla f}(\boldsymbol{x_k})\big) \cdot \boldsymbol{\nabla f}(\boldsymbol{x_k}) \qquad \text{(scalar)}$$

And, using more chain rule, we have the second derivative:

$$\phi''(\alpha) = \left[\mathbf{J}^{\mathbf{T}}(\boldsymbol{\nabla f})\big(\mathbf{x} = \boldsymbol{x_k} - \alpha_k \boldsymbol{\nabla f}(\boldsymbol{x_k})\big) \cdot \boldsymbol{\nabla f}(\boldsymbol{x_k})\right]^T \cdot \boldsymbol{\nabla f}(\boldsymbol{x_k})$$

$$= (\boldsymbol{\nabla f})^T(\boldsymbol{x_k}) \cdot \boldsymbol{H}\big(\boldsymbol{x} = \boldsymbol{x_k} - \alpha_k \boldsymbol{\nabla f}(\boldsymbol{x_k})\big) \cdot \boldsymbol{\nabla f}(\boldsymbol{x_k}) \qquad \text{(scalar)}$$

This formulation in terms of the Hessian and gradient is more useful for implementing a general subroutine: the user will need to supply the function's gradient and Hessian (as a function of the independent variables vector $x_1, x_2, \ldots x_n$, and the subroutine will handle the calculation of ϕ' and ϕ'' automatically based on the derivation above.)

Problem's Specific:

$$f(x, y) = 4x^2 - 4xy + 2y^2 + 8$$

$$\boldsymbol{\nabla f}(\boldsymbol{x}, \boldsymbol{y}) = \begin{bmatrix} 8x - 4y \\ -4x + 4y \end{bmatrix}$$

$$\rightarrow \ \boldsymbol{\nabla f}(\boldsymbol{x} = \boldsymbol{x_k}) = \begin{bmatrix} 8x_k - 4y_k \\ -4x_k + 4y_k \end{bmatrix}$$

$$\boldsymbol{x_k} - \alpha_k \boldsymbol{\nabla f}(\boldsymbol{x_k}) = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - \alpha_k \begin{bmatrix} 8x_k - 4y_k \\ -4x_k + 4y_k \end{bmatrix} = \begin{bmatrix} (1 - 8\alpha_k)x_k + 4\alpha_k\, y_k \\ 4\alpha_k\, x_k + (1 - 4\alpha_k)y_k \end{bmatrix}$$

$$\rightarrow \boldsymbol{\nabla f}\big(\boldsymbol{x} = \boldsymbol{x_k} - \alpha_k \boldsymbol{\nabla f}(\boldsymbol{x_k})\big) = \begin{bmatrix} 8[(1-8\alpha_k)x_k + 4\alpha_k\, y_k] - 4[4\alpha_k\, x_k + (1-4\alpha_k)y_k] \\ -4[(1-8\alpha_k)x_k + 4\alpha_k\, y_k] + 4[4\alpha_k\, x_k + (1-4\alpha_k)y_k] \end{bmatrix}$$
$$= \begin{bmatrix} (8-80\alpha_k)x_k + (-4+48\alpha_k)y_k \\ (-4+48\alpha_k)x_k + (4-32\alpha_k)y_k \end{bmatrix}$$

$$\boldsymbol{J}(\boldsymbol{\nabla f})(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{H}(\boldsymbol{x},\boldsymbol{y}) = \begin{bmatrix} 8 & -4 \\ -4 & 4 \end{bmatrix} \text{ (symmetric, OK)}$$

Since the Hessian is independent of x and y, we have:

$$\boldsymbol{H}\big(\boldsymbol{x_k} - \alpha_k \boldsymbol{\nabla f}(\boldsymbol{x_k})\big) = \boldsymbol{H} = \begin{bmatrix} 8 & -4 \\ -4 & 4 \end{bmatrix}$$

$$\phi'(\alpha_k) = -(\boldsymbol{\nabla f})^{\boldsymbol{T}}\big(\boldsymbol{x_k} - \alpha_k \boldsymbol{\nabla f}(\boldsymbol{x_k})\big) \cdot \boldsymbol{\nabla f}(\boldsymbol{x_k}) = -\begin{bmatrix} (8-80\alpha_k)x_k + (-4+48\alpha_k)y_k \\ (-4+48\alpha_k)x_k + (4-32\alpha_k)y_k \end{bmatrix}^T \begin{bmatrix} 8x_k - 4y_k \\ -4x_k + 4y_k \end{bmatrix}$$
$$= (-80+832\alpha_k)x_k^2 + (96-1024\alpha_k)x_k y_k + (-32+320\alpha_k)y_k^2$$

$$\phi''(\alpha_k) = (\boldsymbol{\nabla f})^{\boldsymbol{T}}(\boldsymbol{x_k}) \cdot \boldsymbol{H} \cdot \boldsymbol{\nabla f}(\boldsymbol{x_k}) = \begin{bmatrix} 8x_k - 4y_k & -4x_k + 4y_k \end{bmatrix} \begin{bmatrix} 8 & -4 \\ -4 & 4 \end{bmatrix} \begin{bmatrix} 8x_k - 4y_k \\ -4x_k + 4y_k \end{bmatrix}$$
$$= \begin{bmatrix} 80x_k - 48y_k & -48x_k + 32y_k \end{bmatrix} \begin{bmatrix} 8x_k - 4y_k \\ -4x_k + 4y_k \end{bmatrix} = 832x_k^2 - 1024x_k y_k + 320y_k^2$$
$$= \frac{d\big(\phi'(\alpha_k)\big)}{d\alpha_k}$$

Given that these calculations are tedious and error-prone polynomials expansions, the result is verified using MATLAB's symbolic toolbox (and with a different approach: ϕ is obtained first, and then the derivative $\frac{d\phi}{d\alpha}$ is taken as a whole):

```
syms alpha real
syms x(alpha) y(alpha)
syms x_k y_k real

f = 4*x^2 - 4*x*y + 2*y^2 + 8;
grad_f = subs(gradient(f,[x,y]),[x, y],[x_k, y_k]);
phi = subs(f,[x,y],[x_k,y_k]-alpha*grad_f');
d_phi = simplify(expand(diff(phi,alpha)));
pretty(d_phi)
```

```
                          2                 2          2          2
96 x_k y_k + 832 alpha x_k  + 320 alpha y_k  - 80 x_k  - 32 y_k

   - 1024 alpha x_k y_k
```

```
dd_phi = simlify(diff(d_phi, alpha));
pretty(dd_phi)
```

```
       2                          2
832 x_k  - 1024 x_k y_k + 320 y_k
```

With ϕ' and ϕ'' obtained, we can finally implement the Steepest Descent algorithm.

Manual Implementation Script (with Intermediate Results Logged)

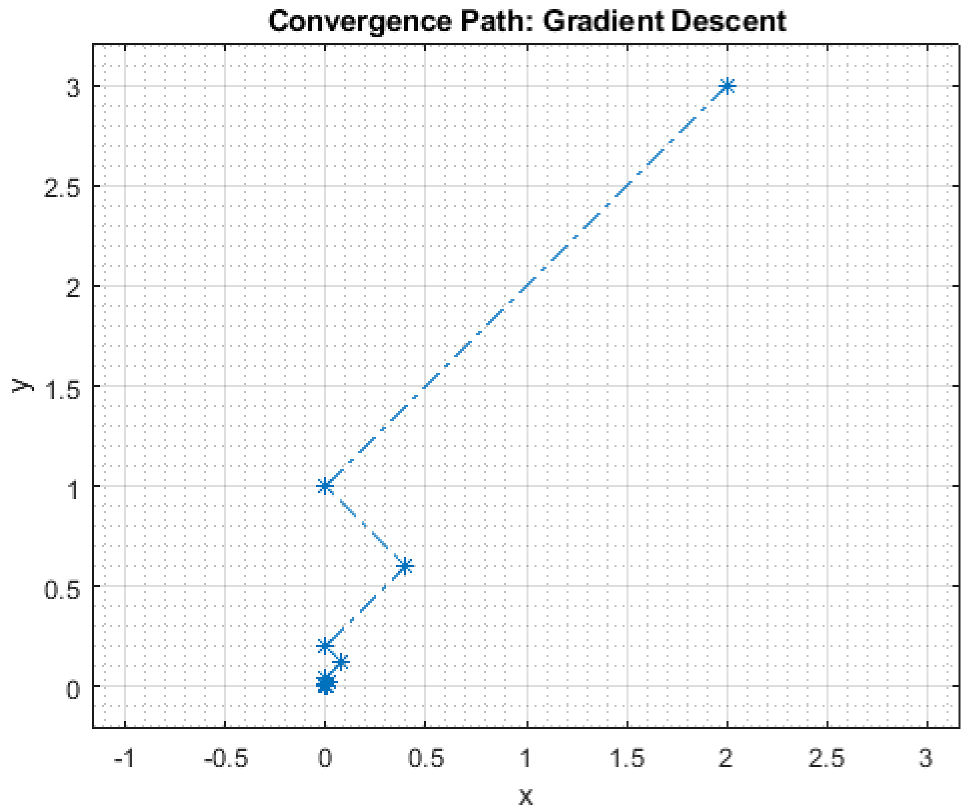
```
alpha_tol = 1e-4; % tolerance for 1D optimization problem
square_norm_err_tol = 1e-6; % Sufficientyly small convergence step's square norm

% phi = objective fuction for 1D optimization problem
f_grad = @(x) [8*x(1) - 4*x(2), -4*x(1) + 4*x(2)]';
f_hess = @(x) [8 -4;-4 4]; % Not needed for manual implementation

d_phi = @(x, alpha) (-80 + 832*alpha)*x(1)^2 + (96 - 1024*alpha)*x(1)*x(2) + (-32 +
320*alpha)*x(2)^2;
dd_phi = @(x) 832*x(1)^2 - 1024*x(1)*x(2) + 320*x(2)^2;
alpha_k_plus_one = @(x, alpha) alpha - d_phi(x, alpha)/dd_phi(x);

prev_guess = [2 3]'; % initial guess
prev_alpha = 0;
iter = 0;
max_iter = 30;

% Iteration Table
iter_num = 1:max_iter;
x_k = zeros(length(iter_num),1);
y_k = zeros(length(iter_num),1);
a_k = zeros(length(iter_num),1);
x_k_plus_one = zeros(length(iter_num),1);
y_k_plus_one = zeros(length(iter_num),1);
err_data = zeros(length(iter_num),1);

while true
    iter = iter + 1;

    while true
        next_alpha = alpha_k_plus_one(prev_guess, prev_alpha);
        if abs(next_alpha - prev_alpha) < alpha_tol
            break;
        end
        prev_alpha = next_alpha;
    end

    next_guess = prev_guess - next_alpha*f_grad(prev_guess);
    err = my2Norm(next_guess - prev_guess);

    x_k(iter) = prev_guess(1);
    y_k(iter) = prev_guess(2);
    a_k(iter) = next_alpha;
    x_k_plus_one(iter) = next_guess(1);
    y_k_plus_one(iter) = next_guess(2);
    err_data(iter) = err;


    if err < square_norm_err_tol
        iter_table = table(iter_num(1:iter)', x_k(1:iter), y_k(1:iter), a_k(1:iter),
x_k_plus_one(1:iter), y_k_plus_one(1:iter),err_data(1:iter), ...
```

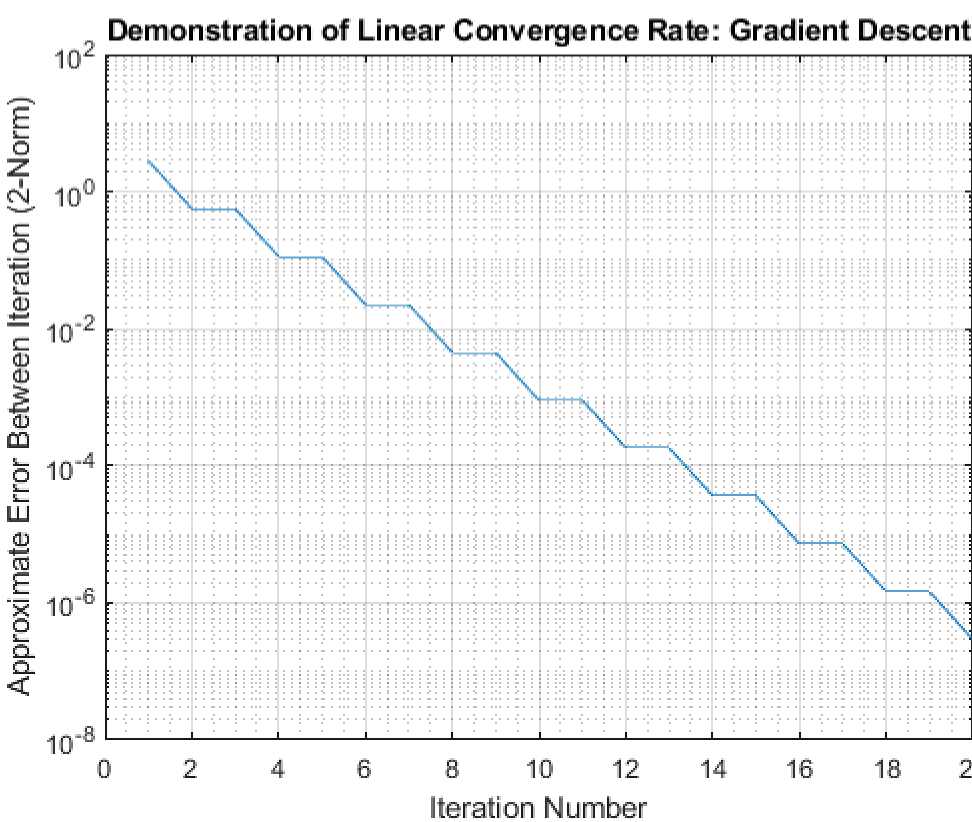
```
            VariableNames = ["Iter.", "x_k","y_k","a_k", "x_k+1", "y_k+1","error"]);
        disp(iter_table);
        break;
    elseif iter >= max_iter
        error("Convergence Failure!");
    end
    prev_guess = next_guess;
end

figure;
plot(iter_table.x_k, iter_table.y_k, "-.*");
xlabel("x");
ylabel("y");
grid on; grid minor; axis padded; axis equal
title("Convergence Path: Gradient Descent");

figure;
semilogy(iter_table.(1), iter_table.error);
xlabel("Iteration Number");
ylabel("Approximate Error Between Iteration (2-Norm)");
title("Demonstration of Linear Convergence Rate: Gradient Descent");
grid on; grid minor;
```

Results:

Iter.	x_k	y_k	a_k	x_k+1	y_k+1	error
1.0000e+00	2.0000e+00	3.0000e+00	5.0000e-01	0.0000e+00	1.0000e+00	2.8284e+00
2.0000e+00	0.0000e+00	1.0000e+00	1.0000e-01	4.0000e-01	6.0000e-01	5.6569e-01
3.0000e+00	4.0000e-01	6.0000e-01	5.0000e-01	-2.6090e-15	2.0000e-01	5.6569e-01
4.0000e+00	-2.6090e-15	2.0000e-01	1.0000e-01	8.0000e-02	1.2000e-01	1.1314e-01
5.0000e+00	8.0000e-02	1.2000e-01	5.0000e-01	2.9143e-16	4.0000e-02	1.1314e-01
6.0000e+00	2.9143e-16	4.0000e-02	1.0000e-01	1.6000e-02	2.4000e-02	2.2627e-02
7.0000e+00	1.6000e-02	2.4000e-02	5.0000e-01	2.1858e-16	8.0000e-03	2.2627e-02
8.0000e+00	2.1858e-16	8.0000e-03	1.0000e-01	3.2000e-03	4.8000e-03	4.5255e-03
9.0000e+00	3.2000e-03	4.8000e-03	5.0000e-01	6.5919e-17	1.6000e-03	4.5255e-03
1.0000e+01	6.5919e-17	1.6000e-03	1.0000e-01	6.4000e-04	9.6000e-04	9.0510e-04
1.1000e+01	6.4000e-04	9.6000e-04	5.0000e-01	1.6371e-17	3.2000e-04	9.0510e-04
1.2000e+01	1.6371e-17	3.2000e-04	1.0000e-01	1.2800e-04	1.9200e-04	1.8102e-04
1.3000e+01	1.2800e-04	1.9200e-04	5.0000e-01	5.1771e-18	6.4000e-05	1.8102e-04
1.4000e+01	5.1771e-18	6.4000e-05	1.0000e-01	2.5600e-05	3.8400e-05	3.6204e-05
1.5000e+01	2.5600e-05	3.8400e-05	5.0000e-01	9.8256e-19	1.2800e-05	3.6204e-05
1.6000e+01	9.8256e-19	1.2800e-05	1.0000e-01	5.1200e-06	7.6800e-06	7.2408e-06
1.7000e+01	5.1200e-06	7.6800e-06	5.0000e-01	3.1171e-19	2.5600e-06	7.2408e-06
1.8000e+01	3.1171e-19	2.5600e-06	1.0000e-01	1.0240e-06	1.5360e-06	1.4482e-06
1.9000e+01	1.0240e-06	1.5360e-06	5.0000e-01	5.6328e-20	5.1200e-07	1.4482e-06
2.0000e+01	5.6328e-20	5.1200e-07	1.0000e-01	2.0480e-07	3.0720e-07	2.8963e-07

```
The numerical solution (x,y) to the local minimization algorithm is:
next_guess =

   2.0480e-07
   3.0720e-07

The corresponding minimized value of the objective function f(x,y) is:
ans =

   8.0000e+00
```


Convergence Path: Gradient Descent
y
x
3
2.5
2
1.5
1
0.5
0
-1
-0.5
0
0.5
1
1.5
2
2.5
3


Demonstration of Linear Convergence Rate: Gradient Descent
Approximate Error Between Iteration (2-Norm)
Iteration Number
10^{2}
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
0
2
4
6
8
10
12
14
16
18
20

The exact same result (up to 11-12 decimal places) was achieved (with identical tolerance parameters) using the generically applicable subroutine (although intermediary logging and visualization are not implemented in the subroutine).

Gradient Descent Subroutine Implementation and Result Comparison

```
function minimum = myGradientDescent(guess, grad, hess, tol)

ALPHA_TOL = 1e-4;
MAX_ITER = 30;

d_phi = @(vector, a) -grad(vector-a*grad(vector))'*grad(vector);
dd_phi = @(vector, a) grad(vector)'*hess(vector - a*grad(vector))*grad(vector);

iter_count = 0;
a_k_guess_prev = 0;
minimum_k_minus_one = guess;

while true
    iter_count = iter_count + 1;

    while true
        a_k = a_k_guess_prev -
d_phi(minimum_k_minus_one,a_k_guess_prev)/dd_phi(minimum_k_minus_one,a_k_guess_prev);
        if abs(a_k - a_k_guess_prev) < ALPHA_TOL
            break;
        end
        a_k_guess_prev = a_k;
    end

    minimum = minimum_k_minus_one - a_k*grad(minimum_k_minus_one);
    err = my2Norm(minimum - minimum_k_minus_one);

    if err < tol
        return
    elseif iter_count >= MAX_ITER
        error("Convergence Failure!");
    end

    minimum_k_minus_one = minimum;
end
end
```

```
subroutine_result = myGradientDescent([2 3]',f_grad,f_hess,square_norm_err_tol)
```

```
subroutine_result =

   2.0480e-07
   3.0720e-07
```

Problem 2:

Approach:

To successfully implement the Newton-Raphson method, we need to obtain the expression for the Jacobian of the vector field. Performing the partial differentiation is tedious and error prone. Therefore, we use MATLAB's symbolic toolbox.

Obtaining the Jacobian

```
function J = myJacobian(F_expr, sym_vector)
sym_N = length(sym_vector);

J = sym(zeros(sym_N,sym_N));

for f_index = 1:sym_N
    for sym_index = 1:sym_N
        J(f_index, sym_index) = simplify(diff(F_expr(f_index),
sym_vector(sym_index)));
    end
end

end
```

```
syms x y z real

f = -4*x + 5*sin(y) + 0.1*z - 5;
g = x^2 + 2*y + exp(-0.5*z) - 5;
h = x + y + z^2 - 12;
F = [f ; g; h];
sym_vector = [x y z]';
J = myJacobian(F,sym_vector)

J =

[ -4, 5*cos(y),         1/10]
[2*x,        2, -exp(-z/2)/2]
[  1,        1,          2*z]

MATLAB_jacobian = jacobian(F,sym_vector)

MATLAB_jacobian =

[ -4, 5*cos(y),         1/10]
[2*x,        2, -exp(-z/2)/2]
[  1,        1,          2*z]
```

Applying the Newton Method:

```
function [guess, error_log, traj] = myNewtonMultiDim(guess, f , jac, tol)

    MAX_ITER = 50;
    dim = length(guess);
    iter_count = 0;
    prev_guess = guess;

    error_log_temp = zeros(1,MAX_ITER);
    traj_temp = zeros(dim,MAX_ITER);

    while true
        iter_count = iter_count + 1;

        currentJ = jac(prev_guess);
        currentf = f(prev_guess);
        update_vector = GaussElimination_PP(currentJ, -currentf);
        guess = prev_guess + update_vector;
        norm = my2Norm(f(guess));

        error_log_temp(iter_count) = norm;
        traj_temp(:,iter_count) = guess;

        if norm < tol
            error_log = error_log_temp(1:iter_count);
            traj = traj_temp(:,1:iter_count);
            return
        elseif iter_count >= MAX_ITER
            error_log = error_log_temp(1:iter_count);
            traj = traj_temp(:,1:iter_count);
            warning("Failure to Converge After Maximum Iteration")
            return;
        end
        prev_guess = guess;
    end

end
```

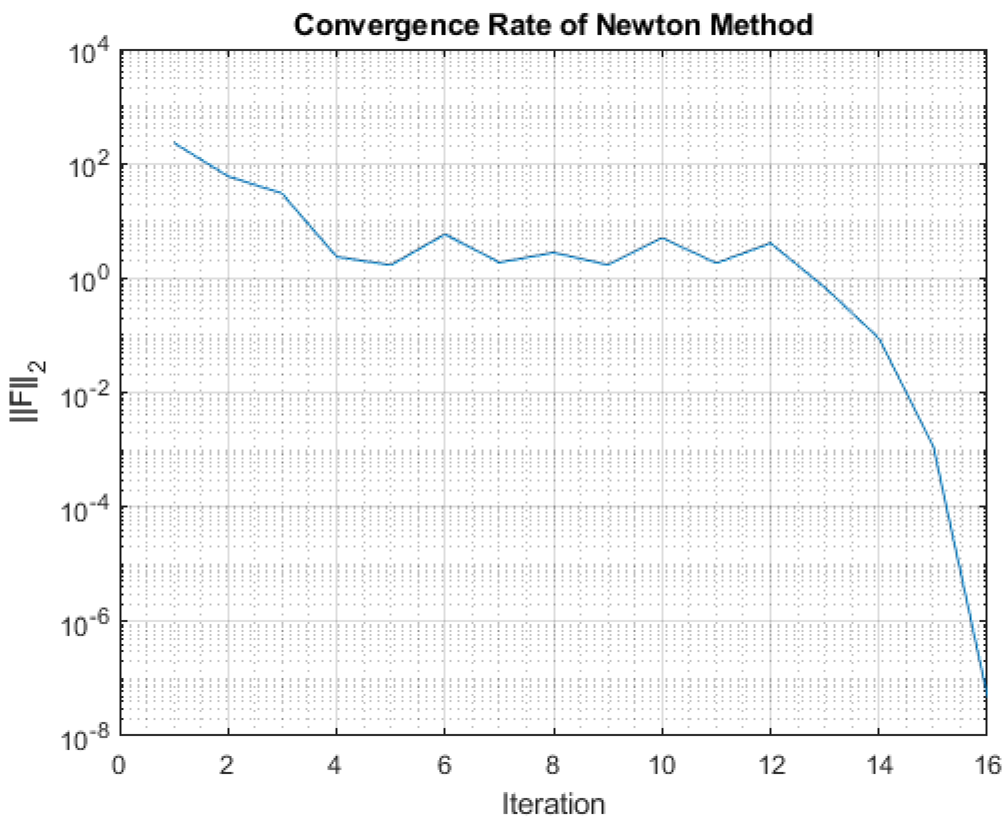
```
f = @(x) -4*x(1) + 5*sin(x(2)) + 0.1*x(3) - 5;
g = @(x) x(1)^2 + 2*x(2) + exp(-0.5*x(3)) - 5;
h = @(x) x(1) + x(2) + x(3)^2 - 12;
F = @(x) [f(x);g(x);h(x)];
J = @(x) [ -4, 5*cos(x(2)),          1/10;
          2*x(1),        2, -exp(-x(3)/2)/2;
            1,        1,          2*x(3)];

init = [0, 0, 0]';

[my_result, error_data, conv_trajectory] = myNewtonMultiDim(init,F,J, 1e-3);
my_result
```

```
my_result =

  -2.8669e-01
   2.3555e+00
   3.1514e+00
```

```
verification = F(my_result)
```

```
verification =

  -1.0636e-08
   4.6804e-08
   2.0314e-09
```

```
figure;
semilogy(error_data);
xlabel("Iteration");
ylabel("||F||_2");
title("Convergence Rate of Newton Method");
grid on; grid minor
```



Appendix – MATLAB Code

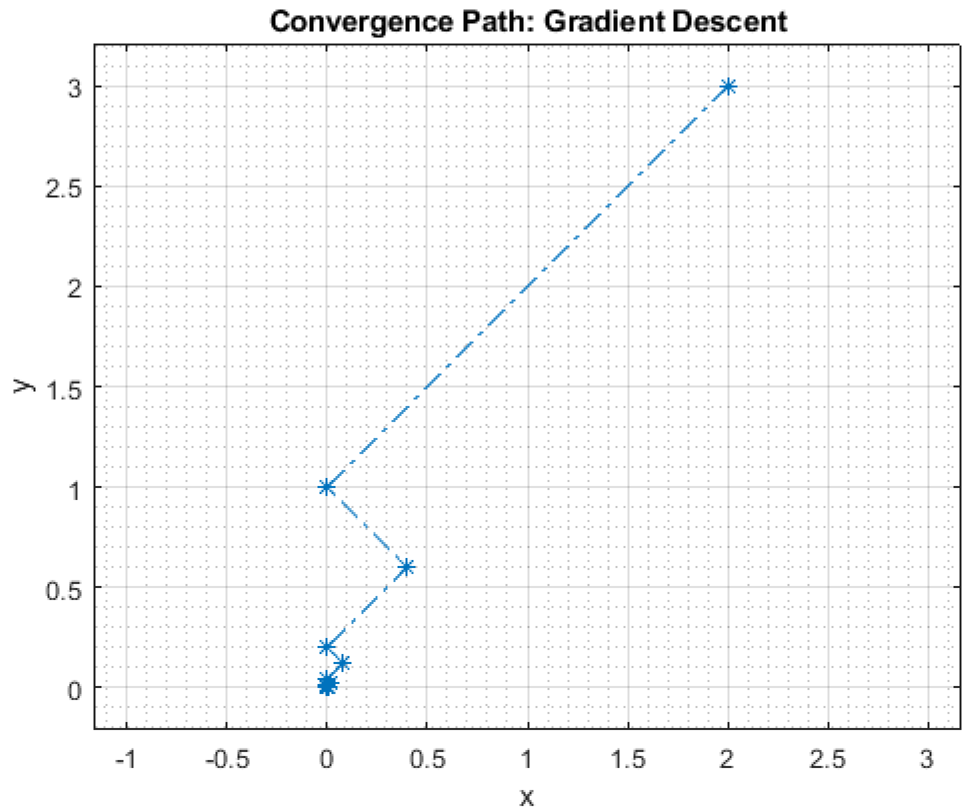
Scripts

```
clc; clear; close all; format shortE
alpha_tol = 1e-4; % tolerance for 1D optimization problem
square_norm_err_tol = 1e-6; % Sufficientyly small convergence step's square norm

% phi = objective fuction for 1D optimization problem
f = @(x) 4*x(1)^2 -4*x(1)*x(2) + 2*x(2)^2 + 8;
f_grad = @(x) [8*x(1) - 4*x(2), -4*x(1) + 4*x(2)]';
f_hess = @(x) [8 -4;-4 4]; % Not needed for manual implementation

d_phi = @(x, alpha) (-80 + 832*alpha)*x(1)^2 + (96 - 1024*alpha)*x(1)*x(2) + (-32 +
320*alpha)*x(2)^2;
dd_phi = @(x) 832*x(1)^2 - 1024*x(1)*x(2) + 320*x(2)^2;
alpha_k_plus_one = @(x, alpha) alpha - d_phi(x, alpha)/dd_phi(x);

prev_guess = [2 3]'; % initial guess
prev_alpha = 0;
iter = 0;
max_iter = 30;

% Iteration Table
iter_num = 1:max_iter;
x_k = zeros(length(iter_num),1);
y_k = zeros(length(iter_num),1);
a_k = zeros(length(iter_num),1);
x_k_plus_one = zeros(length(iter_num),1);
y_k_plus_one = zeros(length(iter_num),1);
err_data = zeros(length(iter_num),1);

while true
    iter = iter + 1;

    while true
        next_alpha = alpha_k_plus_one(prev_guess, prev_alpha);
        if abs(next_alpha - prev_alpha) < alpha_tol
            break;
        end
        prev_alpha = next_alpha;
    end

    next_guess = prev_guess - next_alpha*f_grad(prev_guess);
    err = my2Norm(next_guess - prev_guess);

    x_k(iter) = prev_guess(1);
    y_k(iter) = prev_guess(2);
    a_k(iter) = next_alpha;
    x_k_plus_one(iter) = next_guess(1);
    y_k_plus_one(iter) = next_guess(2);
    err_data(iter) = err;


    if err < square_norm_err_tol
        iter_table = table(iter_num(1:iter)', x_k(1:iter), y_k(1:iter), a_k(1:iter),
x_k_plus_one(1:iter), y_k_plus_one(1:iter),err_data(1:iter), ...
```

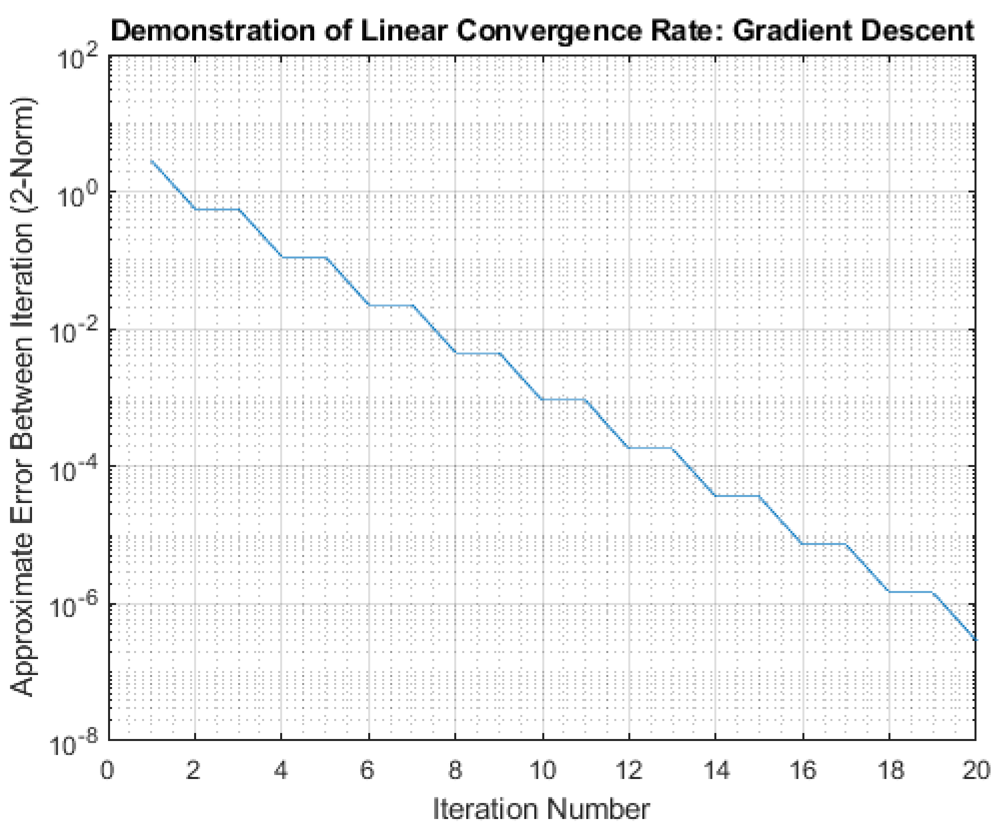
```
            VariableNames = ["Iter.", "x_k","y_k","a_k", "x_k+1", "y_k+1","error"]);
        disp(iter_table);
        break;
    elseif iter >= max_iter
        error("Convergence Failure!");
    end
    prev_guess = next_guess;
end

fprintf("The numerical solution (x,y) to the local minimization algorithm is:");
next_guess
fprintf("The corresponding minimized value of the objective function f(x,y) is:");
f(next_guess)

figure;
plot(iter_table.x_k, iter_table.y_k, "-.*");
xlabel("x");
ylabel("y");
grid on; grid minor; axis padded; axis equal
title("Convergence Path: Gradient Descent");

figure;
semilogy(iter_table.(1), iter_table.error);
xlabel("Iteration Number");
ylabel("Approximate Error Between Iteration (2-Norm)");
title("Demonstration of Linear Convergence Rate: Gradient Descent");
grid on; grid minor;
```

```
      Iter.          x_k           y_k           a_k          x_k+1         y_k+1
error
    __________    ___________    __________    __________    ___________    __________
__________

    1.0000e+00     2.0000e+00    3.0000e+00    5.0000e-01     0.0000e+00    1.0000e+00
2.8284e+00
    2.0000e+00     0.0000e+00    1.0000e+00    1.0000e-01     4.0000e-01    6.0000e-01
5.6569e-01
    3.0000e+00     4.0000e-01    6.0000e-01    5.0000e-01    -2.6090e-15    2.0000e-01
5.6569e-01
    4.0000e+00    -2.6090e-15    2.0000e-01    1.0000e-01     8.0000e-02    1.2000e-01
1.1314e-01
    5.0000e+00     8.0000e-02    1.2000e-01    5.0000e-01     2.9143e-16    4.0000e-02
1.1314e-01
    6.0000e+00     2.9143e-16    4.0000e-02    1.0000e-01     1.6000e-02    2.4000e-02
2.2627e-02
    7.0000e+00     1.6000e-02    2.4000e-02    5.0000e-01     2.1858e-16    8.0000e-03
2.2627e-02
    8.0000e+00     2.1858e-16    8.0000e-03    1.0000e-01     3.2000e-03    4.8000e-03
4.5255e-03
    9.0000e+00     3.2000e-03    4.8000e-03    5.0000e-01     6.5919e-17    1.6000e-03
4.5255e-03
    1.0000e+01     6.5919e-17    1.6000e-03    1.0000e-01     6.4000e-04    9.6000e-04
9.0510e-04
    1.1000e+01     6.4000e-04    9.6000e-04    5.0000e-01     1.6371e-17    3.2000e-04
9.0510e-04
    1.2000e+01     1.6371e-17    3.2000e-04    1.0000e-01     1.2800e-04    1.9200e-04
1.8102e-04
    1.3000e+01     1.2800e-04    1.9200e-04    5.0000e-01     5.1771e-18    6.4000e-05
1.8102e-04
    1.4000e+01     5.1771e-18    6.4000e-05    1.0000e-01     2.5600e-05    3.8400e-05
3.6204e-05
    1.5000e+01     2.5600e-05    3.8400e-05    5.0000e-01     9.8256e-19    1.2800e-05
3.6204e-05
```

```
    1.6000e+01     9.8256e-19    1.2800e-05    1.0000e-01     5.1200e-06    7.6800e-06
7.2408e-06
    1.7000e+01     5.1200e-06    7.6800e-06    5.0000e-01     3.1171e-19    2.5600e-06
7.2408e-06
    1.8000e+01     3.1171e-19    2.5600e-06    1.0000e-01     1.0240e-06    1.5360e-06
1.4482e-06
    1.9000e+01     1.0240e-06    1.5360e-06    5.0000e-01     5.6328e-20    5.1200e-07
1.4482e-06
    2.0000e+01     5.6328e-20    5.1200e-07    1.0000e-01     2.0480e-07    3.0720e-07
2.8963e-07

The numerical solution (x,y) to the local minimization algorithm is:
next_guess =

   2.0480e-07
   3.0720e-07

The corresponding minimized value of the objective function f(x,y) is:
ans =

   8.0000e+00
```



Convergence Path: Gradient Descent
y
x



Demonstration of Linear Convergence Rate: Gradient Descent
Approximate Error Between Iteration (2-Norm)
Iteration Number

```
subroutine_result = myGradientDescent([2 3]',f_grad,f_hess,square_norm_err_tol)
```

```
subroutine_result =

   2.0480e-07
   3.0720e-07
```

Published with MATLAB® R2022a

```
clc; clear; close all;
syms alpha real
syms x(alpha) y(alpha)
syms x_k y_k real

f = 4*x^2 - 4*x*y + 2*y^2 + 8;
grad_f = subs(gradient(f,[x,y]),[x, y],[x_k, y_k]);
phi = subs(f,[x,y],[x_k,y_k]-alpha*grad_f');
d_phi = simplify(expand(diff(phi,alpha)))
pretty(d_phi)
```

```
d_phi(alpha) =

96*x_k*y_k + 832*alpha*x_k^2 + 320*alpha*y_k^2 - 80*x_k^2 - 32*y_k^2 -
1024*alpha*x_k*y_k

                          2                 2         2         2
96 x_k y_k + 832 alpha x_k  + 320 alpha y_k  - 80 x_k  - 32 y_k

   - 1024 alpha x_k y_k
```

```
dd_phi = simplify(diff(d_phi, alpha))
pretty(dd_phi)
```

```
dd_phi(alpha) =

832*x_k^2 - 1024*x_k*y_k + 320*y_k^2

       2                         2
832 x_k  - 1024 x_k y_k + 320 y_k
```

Published with MATLAB® R2022a

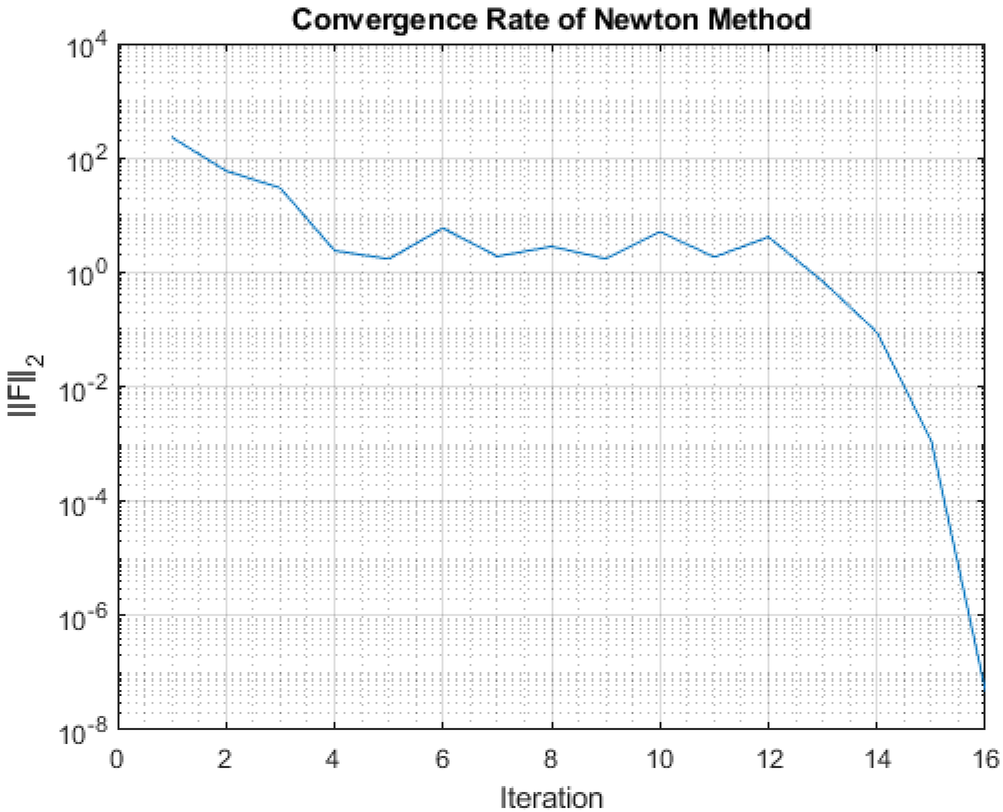
```
clc; clear; close all;
f = @(x) -4*x(1) + 5*sin(x(2)) + 0.1*x(3) - 5;
g = @(x) x(1)^2 + 2*x(2) + exp(-0.5*x(3)) - 5;
h = @(x) x(1) + x(2) + x(3)^2 - 12;
F = @(x) [f(x);g(x);h(x)];
J = @(x) [ -4, 5*cos(x(2)),         1/10;
          2*x(1),        2, -exp(-x(3)/2)/2;
            1,        1,          2*x(3)];

init = [0, 0, 0]';

[my_result, error_data, conv_trajectory] = myNewtonMultiDim(init,F,J, 1e-3);
my_result
```

```
my_result =

  -2.8669e-01
   2.3555e+00
   3.1514e+00
```

```
verification = F(my_result)
```

```
verification =

  -1.0636e-08
   4.6804e-08
   2.0314e-09
```

```
figure;
semilogy(error_data);
xlabel("Iteration");
ylabel("||F||_2");
title("Convergence Rate of Newton Method");
grid on; grid minor
```



Published with MATLAB® R2022a

```
clc; clear; close all;
syms x y z real

f = -4*x + 5*sin(y) + 0.1*z - 5;
g = x^2 + 2*y + exp(-0.5*z) - 5;
h = x + y + z^2 - 12;
F = [f ; g; h];
sym_vector = [x y z]';
J = myJacobian(F,sym_vector)
```

```
J =

[ -4, 5*cos(y),         1/10]
[2*x,        2, -exp(-z/2)/2]
[  1,        1,          2*z]
```

```
MATLAB_jacobian = jacobian(F,sym_vector)
```

```
MATLAB_jacobian =

[ -4, 5*cos(y),         1/10]
[2*x,        2, -exp(-z/2)/2]
[  1,        1,          2*z]
```

Published with MATLAB® R2022a

Subroutine

```
function minimum = myGradientDescent(guess, grad, hess, tol)

ALPHA_TOL = 1e-4;
MAX_ITER = 30;

d_phi = @(vector, a) -grad(vector-a*grad(vector))'*grad(vector);
dd_phi = @(vector, a) grad(vector)'*hess(vector - a*grad(vector))*grad(vector);

iter_count = 0;
a_k_guess_prev = 0;
minimum_k_minus_one = guess;

while true
    iter_count = iter_count + 1;

    while true
        a_k = a_k_guess_prev - 
d_phi(minimum_k_minus_one,a_k_guess_prev)/dd_phi(minimum_k_minus_one,a_k_guess_prev);
        if abs(a_k - a_k_guess_prev) < ALPHA_TOL
            break;
        end
        a_k_guess_prev = a_k;
    end

    minimum = minimum_k_minus_one - a_k*grad(minimum_k_minus_one);
    err = my2Norm(minimum - minimum_k_minus_one);

    if err < tol
        return
    elseif iter_count >= MAX_ITER
        error("Convergence Failure!");
    end

    minimum_k_minus_one = minimum;
end
end

function squared_norm = my2Norm(a)
% my2Norm: dot a vector with itself and take the square root
    squared_norm = sqrt(myInnerProduct(a,a));
end

function inner_product = myInnerProduct(a,b)
% myInnerProduct: Calculate the inner product between two 1D arrays
    n = length(a);
    inner_product = 0;
    for index = 1:n
        inner_product = inner_product + a(index)*b(index);
    end
end
```

```
function J = myJacobian(F_expr, sym_vector)
```

```
sym_N = length(sym_vector);

J = sym(zeros(sym_N,sym_N));

for f_index = 1:sym_N
    for sym_index = 1:sym_N
        J(f_index, sym_index) = simplify(diff(F_expr(f_index),
sym_vector(sym_index)));
    end
end

end
```

```
function [guess, error_log, traj] = myNewtonMultiDim(guess, f , jac, tol)

    MAX_ITER = 50;
    dim = length(guess);
    iter_count = 0;
    prev_guess = guess;

    error_log_temp = zeros(1,MAX_ITER);
    traj_temp = zeros(dim,MAX_ITER);

    while true
        iter_count = iter_count + 1;

        currentJ = jac(prev_guess);
        currentf = f(prev_guess);
        update_vector = GaussElimination_PP(currentJ, -currentf);
        guess = prev_guess + update_vector;
        norm = my2Norm(f(guess));

        error_log_temp(iter_count) = norm;
        traj_temp(:,iter_count) = guess;

        if norm < tol
            error_log = error_log_temp(1:iter_count);
            traj = traj_temp(:,1:iter_count);
            return
        elseif iter_count >= MAX_ITER
            error_log = error_log_temp(1:iter_count);
            traj = traj_temp(:,1:iter_count);
            warning("Failure to Converge After Maximum Iteration")
            return;
        end
        prev_guess = guess;
    end

end

function x = GaussElimination_PP(A,b)

% Gaussian Elimination with Partial Pivoting
n = size(A,1);
```

```
zero_tol = 1e-8; % Cannot use == to compare floating point number

for pivot_index = 1:n % Pivoting from column 1 to column n

    [max_pivot_abs, max_pivot_relative_index] =
myAbsMax(A(pivot_index:end,pivot_index));
    max_pivot_absolute_index = max_pivot_relative_index + pivot_index - 1;

    if max_pivot_abs < zero_tol
        continue
    end

    if max_pivot_absolute_index ~= pivot_index
        A([pivot_index, max_pivot_absolute_index],:) =
A([max_pivot_absolute_index,pivot_index],:);
        b([pivot_index, max_pivot_absolute_index]) =
b([max_pivot_absolute_index,pivot_index]);
    end

    pivot = A(pivot_index,pivot_index);

    for elim_row_index = (pivot_index+1):n % start eliminating rows below the current
pivot

        multiplier = A(elim_row_index,pivot_index)/pivot; % calculating multiplier

        A(elim_row_index,pivot_index) = 0; %% Save 1 calculation, we already know
that this should be zero

        for elim_col_index = (pivot_index+1):n
            A(elim_row_index,elim_col_index) = A(elim_row_index,elim_col_index) -
A(pivot_index,elim_col_index)*multiplier;
        end

        b(elim_row_index) = b(elim_row_index) - b(pivot_index)*multiplier;

    end

end

% disp("DEBUG: Partial Pivoting U =")
% A

x = myBackSubstitution(A,b);

end

function x = myBackSubstitution(A_upper,b)
% myBackSubstitution: personal implementation of back-substitution

    n = length(b); % Calculate Working Dimension
    x = zeros(n,1); % Allocation for Result Vector
    for subStep = n:-1:1 % Reverse Indexing From n to 1
        % Residual = Dot product of sub-vector on the right of the diagonal
        % entry and the sub-vector of known x entries
```

```
        residual = myInnerProduct(A_upper(subStep,subStep+1:n),x(subStep+1:n));
        % x at the current row = b at the current row - residual, then
        % divided by diagonal entry
        x(subStep) = (b(subStep)-residual)/A_upper(subStep,subStep);
    end
end

function [maxVal, maxIndex] = myAbsMax(number_array)
% Given a 1D array, find the number with the maximum magnitude and its
% index inside the array
maxVal = -1;
maxIndex = -1;
for numberIndex = 1:length(number_array)
    current = abs(number_array(numberIndex));
    if current > maxVal
        maxVal = current;
        maxIndex = numberIndex;
    end
end
end

function squared_norm = my2Norm(a)
% my2Norm: dot a vector with itself and take the square root
    squared_norm = sqrt(myInnerProduct(a,a));
end

function inner_product = myInnerProduct(a,b)
% myInnerProduct: Calculate the inner product between two 1D arrays
    n = length(a);
    inner_product = 0;
    for index = 1:n
        inner_product = inner_product + a(index)*b(index);
    end
end
```